Improving the Performance of MPI Derived Datatypes*

William D. Gropp, Ewing Lusk, and Debbie Swider
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Abstract

The Message Passing Interface (MPI) standard provides a powerful mechanism for describing non-contiguous memory locations: derived datatypes. In addition, MPI derived datatypes have a key role in the MPI-2 I/O operations. In principle, MPI derived datatypes allow a user to more efficiently communicate noncontiguous data (for example, strided data) because the MPI implementation can move the data without any intermediate copies to or from a contiguous buffer. In practice, however, few MPI implementations provide support for datatypes that performs better than what the user can achieve by manually packing and unpacking the data into contiguous buffers before calling MPI routines with contiguous memory regions. We develop a taxonomy of MPI datatypes according to their memory reference patterns and show how to efficiently implement these patterns. The effectiveness of this approach is illustrated on a variety of platforms.

1 Introduction

The Message Passing Interface (MPI) standard [4, 1] provides a powerful and general way of describing arbitrary collections of data in memory. Special cases allow users to easily define common cases such as strided data (`MPI_Type_vector`) and indexed data (`MPI_Type_indexed`). Careful modification of the extent of a datatype provides additional ways to describe regular patterns in memory. Such concise and powerful descriptions are necessary to eliminate unnecessary memory motion; without them, the user must copy any data to be sent to a contiguous buffer, pass that to the send routine, and then unpack the data when it is received. In principle, the use of derived datatypes allows the MPI implementation to provide superior performance over what the user could achieve if messages could only contain contiguous regions of memory. Unfortunately, the performance of programs using MPI datatypes is often poor compared to just letting the user pack and unpack the data. This paper shows how the performance of MPI derived datatypes can be improved by recognizing and optimizing for regular patterns in MPI datatypes.

A number of requirements for processing MPI datatypes must be kept in mind while designing a faster approach. Inside an MPI implementation, large messages are normally broken into smaller pieces; for example, they may be broken into packets with a fixed maximum size or there may be a limit on the maximum message size that can be sent at one time. The ability to break up long messages is also required for the efficient implementation of some collective operations. Thus, one critical requirement is that it must be possible to start and stop the processing of a datatype at nearly arbitrary points. In addition, the design should be modular enough that other MPI vendors can adopt its good features without extensive redesign of their implementations.

Another requirement is a practical one: the new approach must have a small number of cases so as to not be too complicated to implement or maintain. It should efficiently handle common datatypes and patterns of access, as well as common data alignments. Finally, it must handle both the MPI-1 and MPI-2 datatypes; this includes `MPI_Type_create_resized` and `MPI_Type_create_darray`.

In the rest of this paper, we shall assume that we are implementing datatypes for a parallel machine with a single data representation; this allows us to view all data as `MPI_BYTE` and to ignore data conversion issues. Many of the techniques described in this paper can be applied to the heterogenous case, but restricting the discussion to `MPI_BYTE` both simplifies some issues and provides important opportunities for additional optimizations.

The approach taken in this paper is based on observing that MPI datatypes require only a small number of data movement primitives, and that these primitives include not only a block copy (e.g., `memcpy`), but also a small collec-

*This work was supported byt the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

tion of loops that contain block copies and pointer offset operations. Several MPI datatypes may map onto the same primitives. By optimizing for these loops, significant performance improvements in the processing of MPI datatypes can be achieved.

In Section 2 we show the performance of datatypes on a variety of MPI implementations, both MPICH and vendor-optimized, and compare with user-packing/unpacking that does not use the MPI derived datatypes. These results demonstrate both that current implementations can be improved and that the MPI_Type_vector optimization in MPICH, which this paper generalizes, provides a significant performance improvement. In Section 3 we introduce the basic loops out of which the MPI derived datatypes can be built. In Section 4 we discuss some of the implementation issues. Section 5 measures the effect of the new approach.

2 Performance of MPI Datatypes

We can see the need for an improvement in the performance of MPI datatypes by testing the performance of two different representations of a vector and compare them to having the user pack and unpack the vector into contiguous memory. Table 1 shows the results of using

```
MPI_Type_vector( 1000, 1, 24,
                 MPI_DOUBLE,
                 &newtype );
MPI_Type_commit( &newtype );
MPI_Send( buf, 1, newtype, ... );
```

as “Vector,”

```
MPI_Datatype t[2];
MPI_Aint     offset[2];
int          blen[2];
offset[0] = 0;
t[0] = MPI_DOUBLE; blen[0] = 1;
offset[1] = sizeof(double)*24;
t[1] = MPI_UB; blen[1] = 1;
MPI_Type_struct( 2, blen, offset, t,
                 &newtype );
MPI_Type_commit( &newtype );
MPI_Send( buf, 1000, newtype, ... );
```

as “Struct,”, and

```
double tmp[1000]; int i;
for (i=0; i<1000; i++)
  tmp[i] = buf[24*i];
MPI_Send( tmp, 1000, MPI_DOUBLE,
          ... );
```

as “User.” Note that for these systems, it is always more efficient for the user to manually form a contiguous copy of the data and send that than it is to use the MPI derived datatypes. In the case of the more convenient struct form, it is significantly better to avoid the MPI datatype.

Note that for the “vector” case, the portable MPICH implementation [3] is faster than the vendor implementations that were tested. MPICH handles MPI_Type_vector and MPI_Type_hvector specially; the method used is a prototype of the approach described in this paper.

3 Basic Memory Operations

An MPI derived datatype can be represented as a tree whose nodes are other MPI derived datatypes and whose leaves are MPI predefined datatypes. The datatype tree describes a series of memory move operations to be used in moving data from or to a contiguous representation to or from the layout described by the datatype tree. The obvious way to implement MPI datatypes is to build a routine that recursively traverses this tree in a depth-first manner; this is the approach taken by the MPICH implementation. Other approaches are possible; for example, the MPI-F [2] implementation builds a finite automaton representing the operations in the datatype.

In this paper, we describe a finite automaton whose basic operations include three special loops; these loops represent all of the MPI derived datatypes. In essence, we are replacing some subtrees of the datatype tree with special optimized leaf nodes.

An alternate way of looking at this approach follows. For efficiency, we need a compact compilation of the data move instructions specified by the datatype. The key is to find an efficient yet simple representation; because all uses of MPI datatypes contain a repitition count, nested loops naturally occur. A classic technique for improving the performance of nested loops is to do loop merging or fusion (converting several loops into one). Another common technique is to replace general transitions (finite state machine or recursive calls) with special, optimized steps (such as loops).

Let us first consider three basic operations and describe how they relate to the MPI derived datatypes. To simplify the discussion, we consider only the “pack” case (moving data to a contiguous buffer by reading from data laid out according to the MPI datatype). The unpack case is similar, exchanging source and destination. The following subsections define the operatoin pack_datatype.

3.1 Strided

The first operation is the strided copy. This is a block move followed by an relative offset in the source. This is a natural for MPI_Type_vector and for datatypes created with the MPI-2 function MPI_Type_create_resized

Table 1. Performance for several different ways of sending 1000 doubles separated by a stride of 24 doubles on three high-performance systems. Numbers are bandwidths in MB/sec. See text for details.

System	Vendor MPI		MPICH (old)		User (with vendor MPI)
	Vector	Struct	Vector	Struct	
ASCI Blue Pacific IBM SP	3.8	2.9	6.7	1.0	9.5
ASCI Blue Mountain SGI Origin 2000	7.6	3.1	20	2.6	44
ASCI Red Intel TFLOPS	32	3.3			35

(in both cases when the underlying type is contiguous). It is also important for datatypes created with `MPI_Type_struct` and containing an `MPI_UB` (the MPI-1 method for accomplishing nearly the same effect as the MPI-2 function `MPI_Type_create_resized`). We use `memcpy` to represent an optimized block-move operation; in practice, an implementation may optimize for single word moves or use any other technique that achieves high copy rates. We also assume that the source of the data to be moved is pointed at by `base` and the destination buffer is pointed at by `dest`; these pointers are to byte-sized values. In C terms, the operation is

```
src = base;
for (i=0; i<n; i++) {
    memcpy( dest, src, len );
    dest    += len;
    src     += offset;
    }
```

Note that in the general case where the input datatype to `MPI_Type_vector` is not a contiguous datatype, the `memcpy` operation may be replaced with a `pack_datatype` operation.

Note that an important part of the datatype operation, setting the position for the next datatype to begin (based on the *extent* of the datatype), is not included here. The `MPI_Type_vector` and `MPI_Type_create_resized` cases differ here, and by making this final step separate, we can unify these cases.

3.2 Variable Length Indexed

The next operation is the most general and moves blocks of variable length from specific locations in memory. In C terms, it is

```
for (i=0; i<n; i++) {
  memcpy( dest,
          base + offset[i],
          len[i] );
  dest += len[i];
  }
```

where base is the buffer address in a routine such as `MPI_Send` (a typical value of `base` for these operations is `MPI_BOTTOM`, of course). This implements `MPI_Type_indexed` and `MPI_Type_struct`.

3.3 Fixed Length Indexed

An important special case (enshrined in the MPI-2 routine `MPI_Type_create_indexed_block`) has constant lengths; its C code is

```
for (i=0; i<n; i++) {
  memcpy( dest,
          base + offset[i],
          len );
  dest += len;
  }
```

This loop represents a *gather* operation.

3.4 Common Representation of Loops

In summary, there are three basic non-contiguous memory operations:

1. Fixed length block with offset relative to the previous element

2. Variable length block with absolute offset

3. Fixed length block with absolute offset

The fourth combination, variable length block with relative offset, does not occur in MPI.

These three loops are parameterized with the following values:

looptype Which of the three loops

n Loop count (number of separate blocks to move)

len[] Length of each block (array of size n or scalar)

offset[] Offsets (array of size n or scalar)

isleaf Indicates whether move operation is `memcpy` or `pack_ datatype`

extent Final extent

In the case where single words are moved, these loops can be highly optimized; for example, taking advantage of prefetch or non-blocking load instructions.

In addition, when working with complex, derived datatypes, we may also need either a pointer or array of pointers to datatypes. That is, for non-leaf nodes in the tree representing a derived datatype, the operation to apply in the loops above is "pack datatype," not "memcpy", and we need to know what that datatype is. In addition, we may need to know a `blockcount` for the number of instances of a derived datatype (in the leaf case, this information is combined with the datatype's length to compute the `len` value).

4 Grubby Details

In this section we briefly mention some of the issues in obtaining an efficient implementation. None of these techniques are new, and almost certainly are already in use in many MPI implementations. We mention them to illustrate the steps that must be taken to achieve high performance with MPI datatypes. Not all of these are fully implemented yet in MPICH .

4.1 Removing the outer loops

Perhaps the most awkward requirement is the one that requires support for partial traversal of the datatype. This means that implicit recursion cannot be used and must be replaced with an explicit stack representing the current location of the operation within the datatype tree. In addition, it must be possible to pause midway through any of the loops. This is particularly difficult near the root of the tree, where the datatypes are all MPI derived types. Because of this, all of the outer loops (loops over the members of derived datatypes that are not the loops identified in Section 3) must be replaced with the equivalent finite automaton, and the current state of the loop becomes part of the state stack that is used to stop and resume processing of a datatype.

4.2 Stack Overflow Tests

Stack overflow tests can be eliminated by placing them in the type creation or type commit operations. This has the advantage of detecting a datatype that is too complex at the time the type is created rather than when it is used.

Note that the SGI implementation of MPI has a type complexity environment variable `MPI_TYPE_DEPTH`, though without a precise definition of what the limit refers to.

4.3 Datatype Layout

The definition of the structure that is used internally in the MPI implementation should be defined so that pushing a datatype on the stack is simply a block copy of a portion of the datatype structure. Further, this part of the datatype can be arranged to be aligned on a cache line, allowing the use of wide load and store instructions where available (e.g., double or quad load), reducing the number of instructions required.

4.4 Pop/Push Optimization

The data movement patterns introduced in Section 3 optimize the lowest levels in the datatype tree, but apply only when data is actually being moved. An important case in MPI are vectors of vectors; for example, describing a section of a 3-d array or using the MPI-2 `MPI_Type_create_darray`. In this case, we will be repeatedly pushing and popping intermediate nodes in the datatype tree. For example, written as a loop, we might have something like this:

```
for (i=0; i<n; i++) {
   push( datatype, count, etc.);
   do_operation( datatype );
   pop;
   src += offset;
}
```

We can optimize this loop in the obvious way by hoisting the push/pop operations out of the loop:

```
push(datatype, count, etc.);
for (i=0; i<n; i++) {
   do_operation(datatype)
   update_stack_top;
   src += offset;
}
pop;
```

where `do_operation` applies the appropriate data move operations. This is a general optimization that can be applied whenever only a single data move operation is specified inside the loop; this includes all MPI indexed and vector descriptions, and some but not all struct descriptions.

Table 2. Performance for a resized datatype

		MPICH		SGI	
Type	Size	Time	MB/s	Time	MB/s
Struct	100	0.000056	14.2	0.000204	3.9
User	100	0.000041	19.7	0.000028	28.4
Struct	10000	0.0019	42.9	0.0117	6.8
User	10000	0.00133	60.3	0.00093	86.3

4.5 Taking Advantage of Known Alignment Restrictions

MPI Datatypes must have the same alignment as the types in the language; i.e., if a `double` has eight-byte alignment, then a data item corresponding to an `MPI_DOUBLE` must also have eight-byte alignment. Taking advantage of known data alignment is important; it allows use of wider load and store instructions. Dynamically detecting alignment is also possible, but hoisting this test out of the loop is another important optimization.

4.6 Precompute Packetization

One reason that loops are not chosen as the basic operation is the need of many MPI implementations to move only as much data as will fit within a packet or memory buffer. For systems with fixed-sized memory buffers, this can be handled at the time that the datatype is committed; it is another example of hoisting loop tests (like the stack overflow) out of the loop when the datatype is created.

Another approach, suitable for relatively large buffers, uses a test to check whether the entire loop fits within the buffer; if not, slower but correct code can be used. Systems with small packet sizes may want to consider using an intermediate buffer of larger size to emulate what the user would do without the MPI datatypes (which Section 2 shows to be superior to most current implementations).

4.7 MPI-2 Datatype Representation

The MPI-2 standard requires that the original calling sequences used to create a derived datatype be retained. The downside of converting all MPI derived datatypes to one of the three loop forms is that a separate data structure must be kept to maintain the information needed for the MPI-2 function `MPI_Type_get_contents` (this is not necessary in MPI-1).

5 Experiments

We illustrate the benefits of the datatype approach of this paper with two experiements. In the first, we send C `doubles` with a stride of 16 doubles using a `MPI_Type_struct` containing an `MPI_DOUBLE` and an `MPI_UB`. We compare MPICH against the SGI implementation of MPI on an SGI Origin 2000 with 250 MHz R10000s. The results are shown in Table 2. The rows labeled "Struct" use the MPI derived datatype; the rows labeled "User" have the program use a loop to pack and unpack the data to and from a contiguous buffer. Note that the MPICH "struct" case, while still slower than "user" case, is much faster than the vendor's "struct" case. Further, the current implementation of MPICH does not exploit the option of packing and unpacking directly to and from the communication buffers; instead it allocates a temporary buffer, moves the data into that, and then sends that temporary buffer by copying it to and from the communication buffers. Eliminating this extra copy will improve the performance of MPICH.

The difference in the performance of the "User" rows for MPICH and SGI are due to the greater performance of contiguous send and receive in the SGI implementation of MPI.

The second example illustrates the advantage of the push/pop optimization. Consider the case of sending part of the face of a 3-dimensional cube to another processor. Specifically, let the cube have sizes `(nx,ny,nz)` and the partial face that we want to send is a y-z face of size `(my,mz)`. To construct a datatype describing this face, we need

```
MPI_Type_vector( my, 1, nx,
                 MPI_DOUBLE, &t1 );
MPI_Type_hvector( mz, 1,
     nx * ny * sizeof(double), t1,
     &newtype );
```

The performance of this derived datatype on an SGI Origin 2000 is shown in Table 3. The cube is $200 \times 200 \times 200$ is size and the face is 100×100. The MPICH row shows the performance using the ideas in this paper (but still making the extra copy described above). The SGI line shows the performance of the SGI implementation of MPI. The third row of data, labeled MPICH-old, shows the benefit of the adding the push/pop optimization.

6 Conclusion

We have shown how to divide the many MPI datatypes into a few categories that can be implemented efficiently. Even though our results are preliminary, and do not include all reasonable performance optimizations, we already provide performance with derived datatypes that is nearly as

Table 3. Performance for a vector of vectors datatype

System	Time	MB/s
MPICH	0.0019	42.6
SGI	0.0066	12.2
MPICH-old	0.0031	26.0

good as what a programmer can do when packing and unpacking contiguous buffers "by hand." Further improvements, particularly those that eliminate the extra copies that packing into and out of a contiguous buffer require, should raise the performance to equal or better than what the application programmer can accomplish.

References

[1] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[2] H. Franke, P. Hochschild, P. Pattnaik, J.-P. Prost, and M. Snir. MPI-F: an MPI prototype implementation on IBM SP1. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 43–55. SIAM, 1994.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.